Variables and Outputs

Terraform – Day2 – Module-5

Learning Objectives

- Declare and use Terraform variables correctly
- Understand variable types (string, number, bool, list, map)
- Apply validation to prevent misconfigurations
- Secure secrets using **sensitive = true**
- Use **.tfvars** **for environment-specific configuration**
- Create and consume Terraform outputs
- Follow best practices for production use

The Problem with Hardcoded Values

```
resource "aws_instance" "web_server" {
 ami          = "ami-0c55b159cbfafe1f0"
 instance_type = "t3.micro"
 tags = {
  Environment = "dev"
  Team        = "payments"
 }
}
```

- **Key Problems (Clearer):**
  - Not reusable across Dev, Staging, Prod
  - Difficult to scale or change later
  - Risk of mistakes in copy-paste edits
  - No single source of truth
  - Hard to enforce standards
- **Discussion Questions (More explicit):**
  - What if Prod needs a larger instance?
  - What if AMI changes by region?
  - What if Finance blocks t3.micro in Prod?

What Are Variables in Terraform?

Variables = Input Parameters to Terraform

- Think of them like function parameters in programming:
  - You **define inputs**
  - Terraform **uses them everywhere**
  - Different values → different infrastructure
- **Banking Use Case (Clearer):**
  Same payment system deployed to:
  - Dev → Small, cheap instances
  - Staging → Near-production setup
  - Prod → Large, secure, redundant infra
- 👉 **One codebase, multiple environments = VARIABLES**

Variables in Action (Before vs After)

Without Variables (Bad)

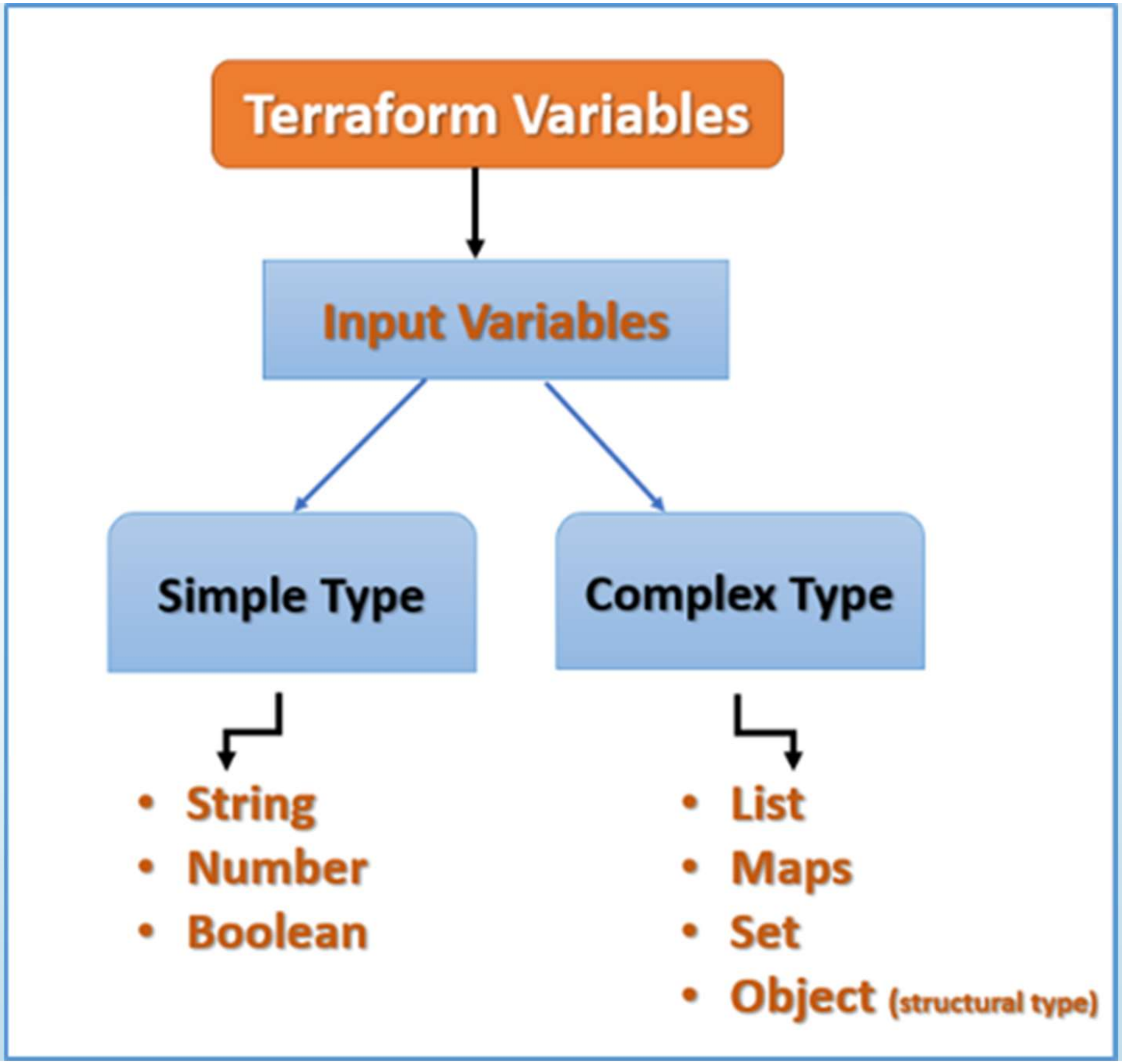
```
provider "aws" {
  region = "us-west-1"
}
resource "aws_instance" "web" {
  instance_type = "t3.micro"
}
```

With Variables (Good)

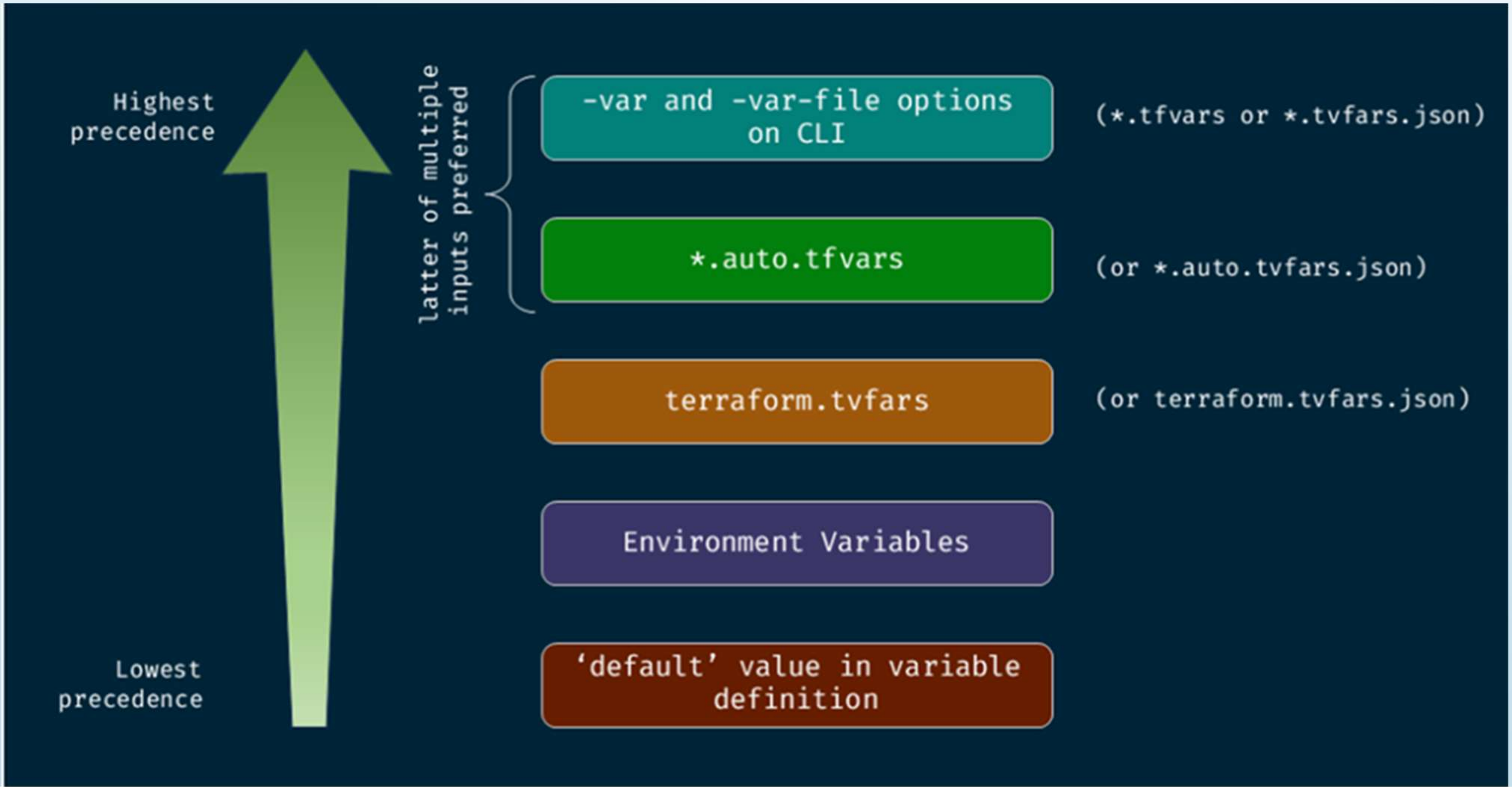
```
provider "aws" {
  region = var.aws_region
}
resource "aws_instance" "web" {
  instance_type = var.instance_type
}

variable “instance_type” {
   default = “t3.micro”
}
```

Types of Variables



Variable precedence



Highest precedence

Lowest precedence

latter of multiple inputs preferred

-var and -var-file options on CLI (*.tfvars or *.tvfars.json)

*.auto.tfvars (or *.auto.tvfars.json)

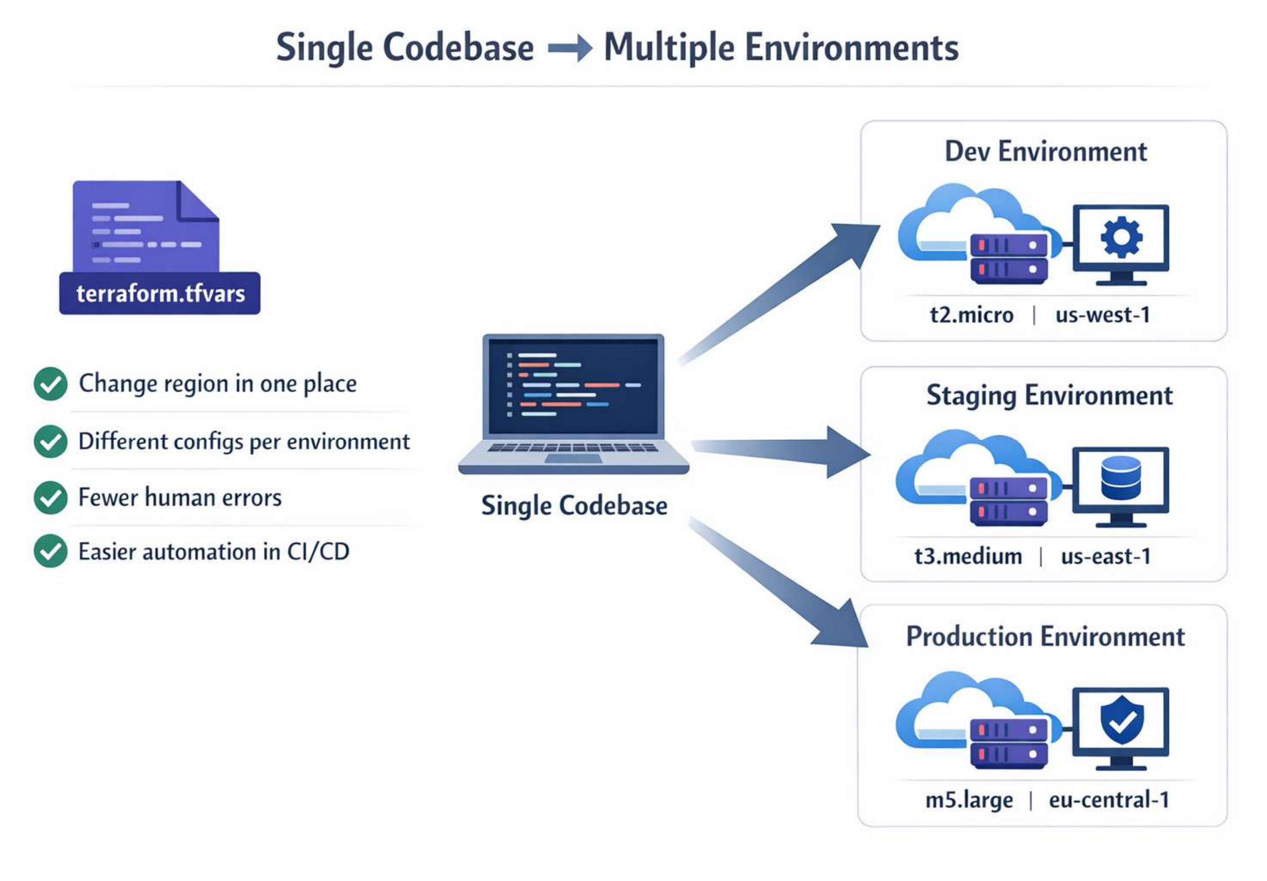
terraform.tvfars (or terraform.tvfars.json)

Environment Variables

'default' value in variable definition

Variables in Action (Before vs After)…

- **Benefits (Stronger wording):**
  - Change region in one place (terraform.tfvars)
  - Different configs per environment
  - Fewer human errors
  - Easier automation in CI/CD



Variable Declaration Syntax

```
variable "variable_name" {
  description = "Human-readable description"
  type        = string
  default     = "optional-default"
  sensitive   = false
  validation {
    condition     = true
    error_message = "Custom error message"
  }
}
```

Variable Declaration Syntax - Example

Example (Clearer)

```
variable "aws_region" {
  description = "AWS region for resources"
  type        = string
  default     = "us-west-1"
}
```

```
provider "aws" {
  region = var.aws_region
}
```

Key Rules:

- Always use `var.<name>`
- Use interpolation when needed:

```
Name = "${var.aws_region}-web-server"
```

Knowledge Check: Variable Syntax

- What is the correct way to reference a variable named **environment** in your Terraform code?
- A) ${environment}
- B) var.environment
- C) variable.environment
- D) env.environment

Correct Answer: B) var.environment

Required vs Optional Variables

Optional (Has Default)

```
variable "aws_region" {
  default = "us-west-1"
}
```

Required (No Default)

```
variable "instance_type"
{
  type = string
}
```

Why this matters:

- Defaults = convenience
- Required = safety (forces explicit decision)

Best Practice:

Use `terraform.tfvars` instead of interactive prompts.

Variable Types: Basic

Type	Meaning	Example
string	Text	`"t3.micro"`
number	Numeric	`3`
bool	True/False	`true`

```
variable "instance_count" {
  type    = number
  default = 1
}

variable "enable_monitoring" {
  type    = bool
  default = false
}
```

Variable Types: Collections

List Example

```
variable "availability_zones" {
  type = list(string)
  default = ["us-west-1a", "us-west-1b"]
}
```

Map Example

```
variable "instance_types" {
  type = map(string)
  default = {
    dev  = "t3.nano"
    prod = "t3.large"
  }
}
```

Variable Validation

```
variable "instance_type" {
  type = string
  validation {
    condition     = contains(["t3.nano", "t3.micro"], var.instance_type)
    error_message = "Only t3.nano or t3.micro are allowed."
  }
}
```

Why Validation Matters:

- Prevents cost overruns
- Enforces standards
- Catches typos early
- Ensures compliance (Banking use case)

Common Validation Functions

Comparison operators - Range checking:

```
validation {
    condition = var.instance_count >= 1 && var.instance_count <= 10
    error_message = "Instance count must be between 1 and 10."
}
```

Banking Context: Use validation to enforce security policies (e.g., production must use multi-AZ, backups enabled).

terraform.tfvars

File: terraform.tfvars

- aws_region = "us-west-1"
- instance_type = "t3.nano"

How it works:

- Automatically loaded by Terraform
- Overrides defaults in `variables.tf`

Environment-Specific Configs

- # dev.tfvars
- instance_type = "t3.nano"

- # staging.tfvars
- instance_type = "t3.micro"

- # prod.tfvars
- instance_type = "t3.large"

Commands:

```
terraform apply
terraform apply -var-file="staging.tfvars"
terraform apply -var-file="prod.tfvars"
```

Before vs After :

- ❌ Three codebases ⟶ messy
- ✅ One codebase ⟶ three config files

Knowledge Check: Variable Validation

You want to ensure the **environment** variable only accepts "dev", "staging", or "prod". Which validation is correct?

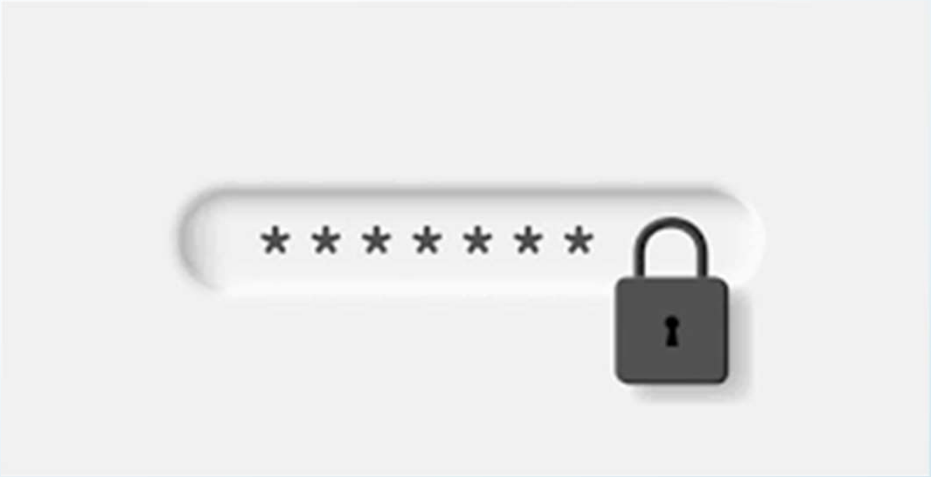
- A) validation { condition = var.environment in ["dev", "staging", "prod"] }
- B) validation { condition = contains(["dev", "staging", "prod"], var.environment) }
- C) validation { condition = var.environment == "dev" || "staging" || "prod" }
- D) validation { condition = length(var.environment) > 0 }

Correct Answer: B)
contains(["dev","staging","prod"], var.environment)

Sensitive Variables

```
variable "db_username" {
  type = string
  sensitive = false
}

variable "db_password" {
  type = string
  sensitive = true
}
```



Effect:

- Masks value in `terraform plan/apply`
- Protects secrets in logs

Sensitive Variables in Action

```
resource "aws_db_instance" "main" {
  username = var.db_username
  password = var.db_password
}
```

Output:

```
password = (sensitive value)
```

Why Important:

- Protects CI/CD logs
- Avoids accidental screen exposure

Important Security Note

⚠️ sensitive = true **only hides output** — it does NOT encrypt state!

Best Practices:

- Use encrypted remote state (S3 + KMS)
- Restrict state access with IAM
- Use AWS Secrets Manager / Vault
- Never commit .tfvars with secrets
- Add *.tfvars to .gitignore

Output Values

```
output "instance_public_ip" {
  value = aws_instance.my_instance.public_ip
}
```

Why Outputs Matter:

- Share values with scripts
- Debug infrastructure
- Document deployments

Viewing Outputs

```
terraform output
terraform output instance_public_ip
terraform output -json
terraform output -raw instance_public_ip
```

Shell example:

```
INSTANCE_IP=$(terraform output -raw instance_public_ip)
ssh ec2-user@$INSTANCE_IP
```

Sensitive Outputs

```
output "db_password" {
  value = var.db_password
  sensitive = true
}
```

Behavior

Behavior:

terraform output → <sensitive>
terraform output -raw → shows actual value

Best Practices

- Use clear variable names
- Always add descriptions
- Group variables logically
- Provide safe defaults
- Never default secrets
- Use validation rules

Key Takeaways

- Variables = flexible infrastructure
- Start with simple types
- Validate everything
- Mark secrets as sensitive

- Use .tfvars for environments
- Outputs power automation
- Never commit secrets